YURT Project Document

Michael Murphy

May 15, 2016

1 Abstract

My semester-long research project in the YURT was split into two parts. For the first part, I worked on developing MinVR2, a virtual reality framework that was to act as an interface between software applications and hardware devices. For the second part, I worked with VRG3D, an existing virtual reality framework, to develop a Shadertoy visualizer. By May, I had learned how YURT-like systems operate and the problems MinVR2 would need to solve in order to overcome the shortcomings of its predecessors, and I had developed a Shadertoy visualizer in VRG3D as a lasting VR artifact.

2 Overview

My original goal for the semester was to transform the Moo Martians game I had written for CSCI-1972 3-D Game Engines into a virtual reality application. This transformation would entail refactoring an existing "flat" graphical program into a VR program, where player interaction with everything from options menus to UFO piloting would be fundamentally different. However, when I learned about alternate projects in the YURT, I reconsidered my initial plan. The project that stood out to me was MinVR2, a framework that was being developed to act as an interface between application developers (like me) and device developers (like Brown's YURT team or Oculus). I realized that I would learn more about YURT-like systems and virtual reality in general by working on the inner workings of the system than by focusing on the development differences from a UX perspective. And after I had helped finish MinVR2, I could use it to port Moo Martians to the YURT.

Unfortunately, MinVR2 did not develop fast enough, and by the second half of March I had to change to another project that could be completed by May. VRG3D was an existing framework, a predecessor to MinVR2 whose shortcomings inspired MinVR2's creation. But VRG3D was functional and came with example projects, so the possibility of finishing a final project with it became more realistic. Given my shortened time frame, I updated my original project from porting over an entire game to rendering Shadertoy demos. All of the steps required for building such a renderer would be required for porting the game, so the idea was that I would still have an interesting project completed even if the rest of Moo Martians couldn't get ported.

3 MinVR2

3.1 Motivation for MinVR2

Virtual reality is fundamentally different from traditional "flat" media, and the reasons extend beyond it being a more immersive experience. Namely, virtual reality is itself a suitcase term for different kinds of experiences ranging from head-mounted displays (HMDs) to virtual rooms such as Brown's YURT and even to less immersive media such as 3-D TVs. While transitioning between traditional flat media such as movie theaters, televisions, and computer monitors simply requires a change of resolution and aspect ratio, transitioning between a limited projection for an HMD and an enveloping projection in a YURT, as well as between alternate forms of input tracking, is a much more involved process.

MinVR2's goal was to act as an interface between creators of applications and creators of devices. Application programmers would need to hook into MinVR2 only once, at which point their program would be able to run on any device that provided a MinVR2 configuration. Device creators would in turn only need to provide one MinVR2 configuration per device.

MinVR2 also sought to fix some of the shortcomings of previous solutions, such as VRG3D. These issues, and how a few were addressed in my project, will be discussed in section 4.

3.2 The Device module

One of MinVR2's major code components was the Device module, and it was this module on which I worked for the first half of the semester. The Device module was to be in charge of organizing the displays and their components into a logical hierarchy. Screens, windows, viewports, stereo vision and screen blending shaders - all components would be represented by nodes in a tree. Device creators would specify a configuration file for their device from which MinVR2 would generate a tree. The tree would be used to synchronize the individual screens for the user, who would then be given access to the computed display information at the application level.

3.2.1 Determining Concurrency Abstraction Scope

A major design question throughout development was determining which pieces of functionality MinVR2 should control and which it should leave for the users to implement. Furthermore, the distinction between purely graphics functionality and VR functionality as a whole was unclear - would a framework that synchronized graphics but not input or other kinds of output such as audio or haptic feedback be meaningful? Furthermore, which abstractions should be reused in the tree representation of the VR system, if any at all?

A central issue was maintaining concurrency between all of the displays, which for the YURT meant concurrency between multiple different computers across a network. The solution was for each computer to run its own deterministic instance of the application, and have information regarding input, graphics, network connectivity, and so forth be centrally coordinated at some point by a master instance. The question in designing the Device module was how to partition this synchronization.

One proposal was to form a synchronization node interface which all nodes would implement. This initially made sense, since every node would need to be synchronized. However, it quickly became clear that while the entire graphics system would need to be synchronized, there was no common interface for it and input, haptic feedback, or other media. The design solution was a scheme that segregated the interfaces. After all, graphics and update mechanisms are separated both at the application layer as well as before they are synchronized by the master network computer, so there was no reason they ever needed to be combined in the Device hierarchy. Furthermore, none of the abstractions for graphics, which was updated on regular intervals, for things like audio or input, which were updated continuously, intermittently, or both.

3.2.2 Determining Control Scope

One of the goals of MinVR2 was for it to be usable by *any* device creator. After choosing to focus on the display side, the key problem was finding the correct abstractions to accomplish the job. Our team consisted of researchers using two different CAVE systems, which gave us meaningful insight into this problem. Namely, the Brown YURT required more layers of abstraction in the Display module than did the Minnesota CAVE. What was appropriately detailed for the YURT appeared redundant for the CAVE, and that level which was appropriate for the CAVE was insufficient for the YURT. Once HMDs were accounted for, the issue became much more immediate.

My design solution was inspired by Scott Meyers' design philosophy for classes - complete and minimal. MinVR2 should solve *existing* problems to the best of our ability, given our collective knowledge of real VR systems. The interface we provide should provide the minimum level of functionality necessary to accommodate all of the devices we planned to support. Redundant abstraction for smaller systems would be acceptable if it was meaningful for more complicated systems. However, abstraction beyond what was necessary for our devices would be inherently fallacious - it is impossible to design for systems that we, by construction, do not know about, therefore nothing we do could possibly have been productive anyway.

3.2.3 Convenience & Public Use

Another concern was how to keep the barrier to entry to MinVR2 as low as possible by providing convenience functionality to the user. The idea was that the less work the user had to do, the more likely they would be to adopt it for their own projects. Such ease of adoption would cause MinVR2 to become a standard solution in the VR industry.

In hindsight, having used VRG3D would have helped me understand the best approach to solving this problem. As will be discussed in section 4, VRG3D suffered precisely because it provided this kind of "convenience" for the user. Namely, when the decisions made by VRG3D were not what the user would have otherwise done, the user's project was rendered impossible without reverse-engineering.

My solution was to make sure that the core functionality was kept separate from implementation-specific functionality. For example, the core of MinVR2 would limit its functionality to strictly that which was entirely platform-independent. From there, everything specific to particular graphics packages (such as OpenGL or Direct3D) would be implemented as user-content that MinVR2 would provide as a separate module. This would solve the desire to lower the barrier to entry for the user without falling into the same anti-pattern that plagued VRG3D.

The key design philosophy is to distinguish what MinVR2 is *coordinating* from what the end-user is *implementing* in a meaningful way. MinVR2 is solving a coordination and synchronization problem for different forms of streaming data - graphics, input, feedback, etc. MinVR2 is *not* providing specific graphics functionality (e.g. video-game output or video streaming) or input functionality (e.g. keyboard input or head tracking). These are tasks performed precisely by the external packages MinVR2 is coordinating and not implementing itself. MinVR2 must provide an interface that allows these dependencies to be pushed to the users, whether they are application developers or device creators, who will attach them to MinVR2 with the appropriately provided hooks.

3.3 The Decision to Switch

Despite the best efforts of the MinVR2 team, the project unfortunately was not progressing at a pace that would ensure its completion within the scope of my semester. As such, I had to change my course of study and devise a project that was doable in the time left in the semester. My time on the MinVR2 development team had taught me about many of the challenges specific to building a VR framework, and I had planned to switch to application development around this time anyway. The only difference was that I would be using VRG3D rather than MinVR2.

4 Shadertoy Visualizer with VRG3D

I switched my project at the end of March from working on MinVR2 to working on a visualizer for shaders. VRG3D was an existing interface that accomplished what MinVR2 was being built to do. VRG3D's shortcomings were the motivation to create MinVR2, but the fact that it both existed and worked won out over those shortcomings. My goal for the remainder of the semester would be to implement a Shadertoy graphics demo visualizer in the YURT.

4.1 Shadertoy

Shadertoy is an online graphics demo website. Users write small graphical applications, or demos, and post them to the site. Specifically, each demo consists of a single fragment shader that is run on a full-screen quad. Rather than render traditional polygon geometry, shaders raymarch creatively designed noise functions to produce fascinating pieces of art. Many of these shaders are fascinating works of art, but they are unfortunately limited to flat screens. I thought standing inside them would be a fun and unique new way to experience them.

4.2 VRG3D And Its Discontents

VRG3D is a VR library that was built on top of G3D, an exsting 3-D graphics library that wrapped OpenGL. The design flaws in VRG3D became apparent as I began using it, and the task of reverse-engineering it to undo these flaws became a prerequisite to writing VR applications, whether they were ports or new ideas. A number of the chief issues are delineated here:

4.2.1 Mixing Implementation with Interface

VRG3D tied itself explicitly to a particular version of G3D, which itself was tied to a particular version of OpenGL. While perhaps a useful expedient move 10 years ago, it ensured that VRG3D would be unable to handle library updates or changes in user-side programming. By failing to abstract out the graphics implementation from the graphics interface, VRG3D was inherently unable to provide portable or reusable functionality or abstraction.

The OpenGL fixed function pipeline (FFP) was deprecated in 2008, and its replacement, the programmable shader pipeline (PSP), had evolved to take its place and since become ever more powerful and flexible. Yet the flaw ran deeper than simply failing to keep up to date with unforeseeable advancements. VRG3D provided a solution to a problem, namely synchronizing multiple instances of a program and calculating the geometric values necessary to display each of them correctly in a virtual reality program. However, VRG3D made unwarranted assumptions about how the user would be *implementing* their own program, which was unrelated to the coordination problem. Rather than provide the valuable information it had calculated to be incorporated into specific implementations, it "conveniently" injected that data into its own implementation.

This design choice manifested itself in concrete problems for me, the user. In order to render my own geometry properly with the new OpenGL pipeline, I had to reverse-engineer the view, projection, and perspective matrices from the data I could scrape from OpenGL and from what I could learn about the implementation of VRG3D. Had I gone further and ported over Moo Martians in its entirety, I would have had to keep track of which framebuffers VRG3D was implicitly assuming I rendered into, as well as other likely OpenGL setting. This was a problem (in fact a class of problems) that arose from forcing implementation details onto clients. In other words, even if OpenGL had never evolved, this design problem would still have persisted.

4.2.2 Many External Dependencies

While external dependencies are not bad in and of themselves, they increase the difficulty with which any given project can be (re)used. VRG3D's reliance on many specific libraries, whose support had in some cases long since been discontinued, made reusing it all the more difficult. Specifically, it was essentially impossible to develop remotely. Both the CCV and CS Department filesystems had the legacy dependencies installed and intricate build macros and environment variables in place; without these, chasing down the many dependencies (VRPN, G3D8, FreeImage, etc.) and installing them would incur substantial time costs. Given the pitfalls surrounding VRG3D on friendly territory, developing from home was prohibitively expensive.

4.3 VRG3D's Success

The power VRG3D granted was much greater than that which it took away. Simply enabling users to develop YURT applications at all was priceless. Furthermore, the fact that it has consistently enabled the creation of VR projects for years is a testament to its durability. Specifically, while its dependence on particular versions of graphics libraries made it unportable in one way, the fact that it has been able to run on many dissimilar devices has proven that it is nonetheless very portable in other ways. Given that this kind of portability is the central problem it was designed to solve, it is safe to say that VRG3D has proven far more portable than not. MinVR2 development should focus on what VRG3D accomplished correctly just as much as what it struggled with.

4.4 Using the CCV computers

The CCV computer system, while certainly capable of supporting developers, had a number of drawbacks that made working on it a bit harder than usual. My lack of general knowledge about very particular system settings and solutions, and not knowing what I didn't know, formed a recurring theme. A brief enumeration of the issues is given here:

- **The Module system** The CCV filesystem supports many users whose vast numbers and requirements preclude any notion of a standard version for dependencies. The Modules system manages all of these, but knowing which particular versions to change, add, or remove is unclear.
- **Deprecated graphics drivers** While the YURT and test wall have high-powered graphics cards with updated drivers, none of the development machines had graphics drivers that had been updated in 10 years. This made locally developing modern graphics applications impossible, forcing all development to be done either on the YURT or the test wall.
- **Unforeseen system changes** At one point the operating system was updated, which broke my code and other existing demos. At another time, the VRPN input server was removed from the test rig for a week, which made using the rig impossible.
- **Lack of common developer tools** Qt5 and its IDE, QtCreator, are an industry standard for C++ development, and the CS Department uses them in all its C++-based classes. The filesystem did not have these installed, and I learned that it did not support modern versions of them, either. The time spent installing these tools was highly profitable. Merging codebases without a sufficiently powerful IDE is impractical and a waste of time for both students and researchers. My hope is that future coders will benefit from this upgrade.

The biggest cost incurred by climbing this learning curve came from my dependence on others who knew everything I needed to know. The knowledge was not internet-searchable, nor could it be deduced simply by thinking harder. For example, in a list of hundreds of dependencies, each with possible subversions, which am I to know to use, or suspect to be the cause of or solution to a problem? Trying any given piece of advice takes little time, but waiting for it costs time, and having to constantly give it costs the people helping me time.

While some problems will still require expert advice, a lot of knowledge needs only to be written down to be effectively conveyed. To this end, I have compiled a separate document outlining all of this particular information with the hope that the barrier to entry will be greatly reduced for future YURT developers. The most recent version of this document is maintained on the CCV wiki pages as the CCV Development Guide.

4.5 Shadertoy Successes

After switching projects midway through the semester, weathering all manners of strange delays, navigating a number of interesting software engineering challenges, and puzzling out a bit of geometric magic, I was finally able to produce a working, correct implementation of a Shadertoy visualizer for the YURT. Any demo can be retrofitted into the YURT, and advanced artistic demos from the web need not have any of their internal voodoo parsed. The key task is to modify the raycasting code at the top to incorporate the proper camera and projection matrices from the VR device. Given the example code, future users should hopefully be able to port any 3D Shadertoy experience to the YURT.